Circle 520

Protect Your Data With Forward Error Correction

S.K.SHENOY
Naval Physical & Oceanographic Lab., Thrikkakara PO, Kochi 682 021, India; e-mail: kedar@satyam.net.in

Multiprocessor architectures have become commonplace in embedded applications these days. The processors often are interconnected via simple asynchronous serial links. These links employ UARTs that don't offer any built-in error control other than simple one-bit error detection using parity. Many designers (except for communication specialists) are unaware that it's not difficult to incorporate reasonably powerful error correction in software.

The performance overheads are quite tolerable in most applications. In fact, this error-correction method can be reduced to simple table lookups.

The method described here, based on the well-known Hamming code technique, enables correction of single-bit errors. The method also can be extended to burst errors as explained below. This forward-error-correction method is useful in applications where data integrity is important. In addition, it's beneficial in simplex systems or long transmission delay systems in which error detection, acknowledgements, and retransmission aren't feasible. One good example is an underwater acoustic data-communication system.

The technique illustrated here uses 4-bit data characters. Three parity bits are added to this data character to form the 7-bit character that's transmitted. This character size is very convenient, since it's supported by most UARTs and other communication devices.

Let the Hamming-encoded data bits be numbered as shown below, where B1 is the LSB:

B1 B2 B3 B4 B5 B6 B7
P1 P2 D1 P3 D2 D3 D4

The corresponding bits P_n and D_n indicate how this character is assembled. P1 to P3 are the inserted parity bits, which are computed as shown below, while D1 to D4 are the data bits. Note that the P_n bits occur at positions corresponding to powers of 2. Representing each B_n in terms of B_ms, which correspond to powers of 2, one can write:

B3 = B1 + B2
B5 = B1 + B4
B6 = B2 + B4
B7 = B1 + B2 + B4

B1 occurs in the expansions for B3, B5, and B7; B2 in B3, B6, and B7; and B4 in B5, B6, and B7.

Using this information, the Hamming coding requires that B1 be represented as the parity of bits B3, B5, and B7, and so on. Thus, the parity bits are computed in terms of the data bits as:

P1 = D1 + D2 + D4
P2 = D1 + D3 + D4
P3 = D2 + D3 + D4

where + indicates the Exclusive-OR operation.

On the receive side, the values of the P1, P2, and P3 bits are recomputed and compared with the received values of P1, P2, and P3. The result is then assigned to R1-R3 such that if the received and computed P1 match, then R1 = 0. Otherwise, R1 = 1; likewise for R2 and R3. If R1-3 is treated as a 3-bit number, it will indicate the bit in error as shown *(Table 1)*.

The erroneous bit is then corrected by inverting it. The attached C program implements the technique described *(see the listing, next page)*. Because 8-bit data is typically used, the program takes each 8-bit character, splits it into two 4-bit nibbles, then encodes and transmits each nibble as a 7-bit character. The 8-bit character is reassembled on the receive side after error correction.

This technique also can be extended to correct burst errors. For correction of burst errors up to seven bits in length, seven characters are assembled and stacked as shown to form a block *(Table 2)*.

The vertical slices corresponding to each bit position (A1 to G1) are then taken to form a 7-bit character and transmitted without any further encoding. All seven such characters are thus transmitted. On the receive side,

TABLE 1: BIT-ERROR INDICATION

R3	R2	R1	Receive status
0	0	0	No error
0	0	1	B1 in error
0	1	0	B2 in error
0	1	1	B3 in error
1	0	0	B4 in error
1	0	1	B5 in error
1	1	0	B6 in error
1	1	1	B7 in error

TABLE 2: BLOCK FORMATION

Char 1	A1	A2	A3	A4	A5	A6	A7
Char 2	B1	B2	B3	B4	B5	B6	B7
Char 3	C1	C2	C3	C4	C5	C6	C7
Char 4	D1	D2	D3	D4	D5	D6	D7
Char 5	E1	E2	E3	E4	E5	E6	E7
Char 6	F1	F2	F3	F4	F5	F6	F7
Char 7	G1	G2	G3	G4	G5	G6	G7

```
/*
 * Protect Your Data Using Forward Error Correction
 * S.K.Shenoy, NPOL, Kochi, India
 */

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <dos.h>

typedef unsigned char byte;

/* Returns 1 if ith bit of the character
 * 'value' is set, else returns 0 */
byte    BitSet( byte value, byte i )
{
 byte    bit = 0x1;

 bit = bit << i;
 if ( ( value & bit ) == bit )
  return( 1 );
 else
  return( 0 );
}

/* Sets a specified bit of a character to 1 or 0 */
void SetBit( byte *Val, byte BitPos, byte BitVal )
{
 if ( BitVal == 0 )                          /* Reset bit */
  *Val = *Val & ~( 0x01 << BitPos );
 else
  *Val = *Val | ( 0x01 << BitPos );
}

/* Splits the input 8-bit character into two nibbles, encodes each nibble
   into 7-bit hamming codes by adding the 3 parity bits. */
void GenHammingCodes( byte chr, byte *ch1, byte *ch2, byte CodeTab[ ] )
{
 *ch1 = ( chr & 0x0F );                      /* Get LS Nibble of data */
 *ch2 = ( chr & 0x0F0 ) >> 4;                /* Get MS Nibble of data */
 *ch1 = CodeTab[ *ch1 ];
 *ch2 = CodeTab[ *ch2 ];
}

/* Inverts the specified bit */
void InvertBit( byte *Val, byte BitPos )
{
 if (Bitset( *Val, BitPos ))
  SetBit( Val, BitPos, 0 );
 else
  SetBit( Val, BitPos, 1 );
}

/* Inverts the specified bit */
void CorruptBit( byte *Val, byte BitNo )
{
 InvertBit( Val, BitNo );
}

/* Encodes the input 4-bit value into 7-bit hamming code
   by adding the 3 parity bits. */
byte    ComputeHammingCode( byte chr )
{
 byte    code, p0, p1, p2;

 code = ( chr & 0xF ) << 3;      /* Get LS data Nibble in 4 MS bits
                                    of 7-bit Code word */

 /* Compute the three parity bits */
 p0 = ( BitSet( code, 3 ) ^ BitSet( code, 4 ) ^ BitSet( code, 6 )) & 0x1;
 p1 = ( BitSet( code, 3 ) ^ BitSet( code, 5 ) ^ BitSet( code, 6 )) & 0x1;
 p2 = ( BitSet( code, 4 ) ^ BitSet( code, 5 ) ^ BitSet( code, 6 )) & 0x1;
 /* Insert the parity bits in the 3 LS bits of code word */
 SetBit( &code, 0, p0 );
 SetBit( &code, 1, p1 );
 SetBit( &code, 2, p2 );
 return( code );
}

/* Generate 16-entry Coding Table of 7-bit codes
   for data nibbles 0000 to 1111 */
void GenerateCodeTable( byte CodeTab[ ] )
{
 byte    i;

 for( i = 0; i < 16; i++ )
  CodeTab[ i ] = ComputeHammingCode( i );
}

/* Generates the 128-entry Decode Table of 7-bit codes
   for "corrupted" codes 0000000 to 1111111 */
void GenerateDecodeTable( byte DecodeTab[ ], byte CodeTab[ ] )
{
 byte    i, DataNibble, ComputedCode, CorrectCode, CorruptBitIndex;

 /* i represents the 128 possible 'corrupt' 7-bit codes */
 for( i = 0; i < 128; i++ ) {
  DataNibble = ( i & 0x78 ) >> 3;          /* Get only the 4-bit data nibble */
  /* Get correct code for 'corrupt' code */
  ComputedCode = CodeTab[ DataNibble ];
  CorruptBitIndex = ( i & 0x7 ) ^ ( ComputedCode & 0x7 );
  CorrectCode = i;                                  /* Correct the 'corrupt' code */
  if( CorruptBitIndex ) /* Non-Zero value = the bit number which is corrupt*/
  {
   /* The following if and else if statements compensate for
   the exchange of the positions of P3 and D0 bits in forming
   the character*/
   if( CorruptBitIndex == 3 ) CorruptBitIndex = 4;
   else if( CorruptBitIndex == 4 ) CorruptBitIndex = 3;
   InvertBit( &CorrectCode, CorruptBitIndex-1 );   /* Correct the error */
  }
  DecodeTab[ i ] = CorrectCode;                          /* Fill the table */
 }
}

/* Corrects the errors in the two receive-encoded characters, strips the
   parity bits and re-assembles the original 8-bit character. */
void ReAssembleChar( byte *ch1, byte *ch2, byte *ch, byte DecodeTab[ ] )
{
 byte    c1,c2;
 c1 = DecodeTab[ *ch1 ];          /* Correct the first code word */
 c2 = DecodeTab[ *ch2 ];          /* Correct the second code word */
 /* Combine the two data nibbles */
 *ch = ( ( c1 >> 3 ) & 0xF ) | ( ( c2 << 1 ) & 0xF0 );
}

void main( void )
{
 byte    val = 'a';
 byte    *ch;
 byte    val1, val2;
 int     BitNo;
 byte    CodeTab[ 16 ];
 byte    DecodeTab[ 128 ];

 /* Generate the Encode and Decode tables */
 GenerateCodeTable( CodeTab );
 GenerateDecodeTable( DecodeTab, CodeTab );

 while ( 1 )                                    /* Forever Loop */
 {
  if ( kbhit( ) )                               /* Key pressed */
  {
   val = getch( );                              /* Read user data */
   if (val == 27)
   {
    exit( 0 );                                  /* Exit if Escape key pressed */
   }
   /* Generate the 2 Hamming codes */
   GenHammingCodes( val, &val1, &val2, CodeTab );
   printf( "val = %x val1 = %x val2 = %x\n", val, val1, val2 );

   printf( "Enter bit position to be corrupted ( 0 -> 6 ):" );
   fscanf( stdin, "%d", &BitNo );
   CorruptBit( &val1, (byte) BitNo );           /* Corrupt the specified bit */
   CorruptBit( &val2, (byte) BitNo );
   printf( "val = %x val1 = %x val2 = %x\n", val, val1, val2 );
   /* Correct & re-assemble char */
   ReAssembleChar( &val1, &val2, ch, DecodeTab );
   printf( "ch = %x\n", *ch );
  }
 }
}
```

The error correction consequently is carried out on each reassembled character (Char 1-7).

In the event of burst errors, at most one bit of each character (Char 1 to Char 7) will be corrupted and can be corrected. For this format, data must be available in blocks of seven characters, which may not be the case in many applications. In such instances, the first character, Char 1, can be used to indicate the actual number of data characters that follow in the block. The rest of the characters, which are dummy characters inserted to make up the seven-character block, can be ignored. The block data count also is protected by the error correction. Note that there are a number of ways of marking packet boundaries in asynchronous communication systems.

The demo C program takes the data character input by the user and splits it into two 4-bit nibbles. Each nibble then is encoded by adding three parity the 7-bit hamming code. A single-bit error can be simulated on these codes at a user-specified bit position by inverting the bit concerned. On the receive side, the error is corrected and the original 8-bit character is reassembled from the two data nibbles extracted from the encoded characters.

The encoding table is generated by applying the encoding algorithm to each of the 16 4-bit patterns (0000-1111), resulting in a 16-entry table with each entry containing a 7-bit encoded value. The encoded value for the data nibble xxxx is given by the 7-bit pattern at the xxxx table position.

The decoding table is generated by applying the decoding algorithm to each of the 7-bit patterns (0000000-1111111), representing the possible received values of encoded data. This creates a 128-entry table with 7-bit entry values. If the received bit pattern is xxxxxxx, the 7-bit entry at index xxxxxxx represents the corrected data character.

and then used for all subsequent encoding/decoding activity.

The program was compiled and tested in Borland C 4.5 on a PC. The encode and decode routines can be easily ported to almost any C environment, including cross compilers for microcontrollers. The encoding and decoding tables can be generated and stored in PROM memory.

A note on the program listing: The data and parity bits have been interleaved in the above discussion for simplicity. This isn't a requirement. In the program listing, the four data bits and three parity bits are conveniently packed together as shown below to form the 7-bit data:

B1 B2 B3 B4 B5 B6 B7
P1 P2 P3 D1 D2 D3 D4

The only effect of this rearrangement is that the "Bit In Error" designations shown in Table 1 must be adjusted to indicate the new bit positions.

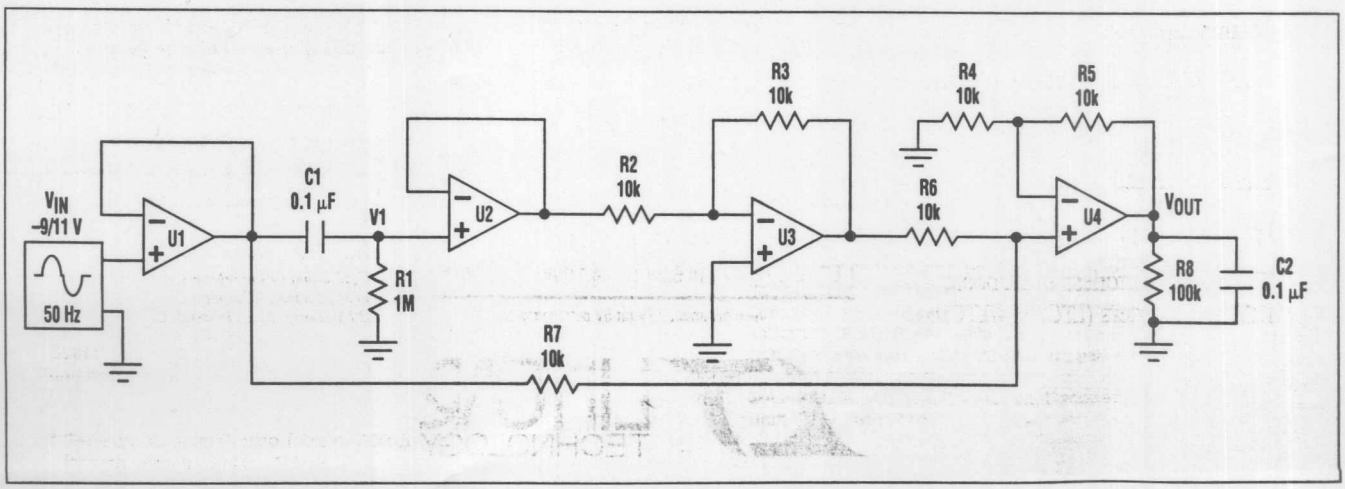
Circle 521

Simple AC-Stop, DC-Pass Circuit Uses Four Op Amps

ANKUR BAL and ABHA JAIN
Delhi College of Engineering, Delhi, India;
e-mail: ankurbal@hotmail.com

The circuit presented can be helpful in applications where the desired output is a dc offset voltage of an ac signal. Such an application would be the design of an electronic watt/watt-hour meter. For example, when calculating active ac power, $V_P*\sin(\omega t)$ (proportional to voltage across the load) and $V_C*\sin(\omega t + \psi)$ (proportional to the current through the load) are multiplied, resulting in $K*V_P*V_C*\cos(\psi)$ [dc offset] and $K*V_P*V_C*\cos(2\omega t + \psi)$ [ac harmonic].

Because the required output signal is just the dc term, the ac term must be eliminated. This type of function can't be performed by conventional low-pass filters. However, the simple and inexpensive circuit shown accomplishes the job *(Fig. 1)*.



1. This simple op-amp circuit is used to extract the dc offset voltage from an input signal containing low-frequency ac powerline components.